Resolution of Interfering Product Fragments in Software Product Line Engineering

Anatoly Vasilevskiy and Øystein Haugen

SINTEF, Pb. 124 Blindern, 0314 Oslo, Norway
{anatoly.vasilevskiy,oystein.haugen}@sintef.no

Abstract. The Common Variability Language (CVL) allows deriving new products in a software product line by substituting fragments (placement) in the base model. Relations between elements of different placement fragments are an issue. Substitutions involving interfering placements may give unexpected and unintended results. However, there is a pragmatic need to define and execute fragments with interference. The need emerges when several diagrams are views of a single model, such as a placement in one diagram and a placement in another diagram reference the same model elements. We handle the issue by 1) classifying interfering fragments, 2) finding criteria to detect them, and 3) suggesting solutions via transformations. We implement our findings in the tooling available for downloading.

Keywords: graph transformations, software product lines, fragment substitutions, adjacent, interference, cvl, conflict resolution

1 Introduction

Software Product Line Engineering (SPLE) [1] has proved itself as a valuable approach to produce configurable and customizable systems. CVL is a domain-

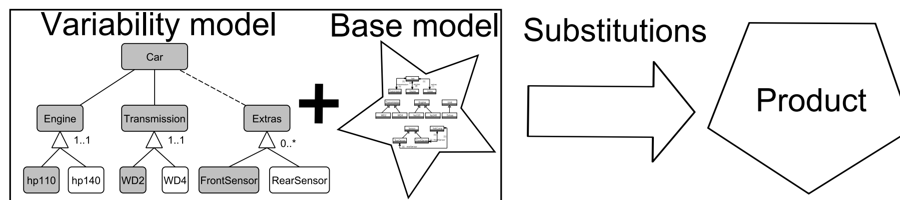
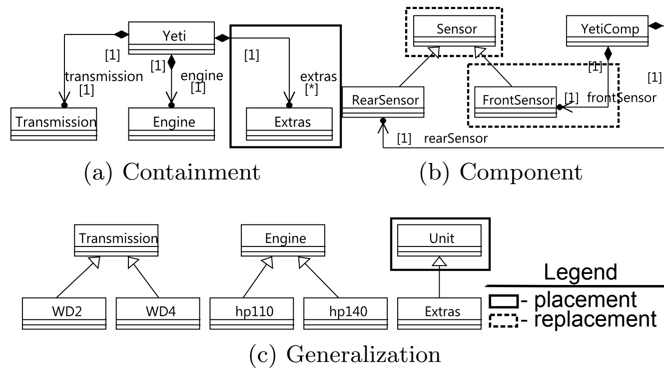


Fig. 1: CVL product derivation workflow

specific language [2,3] for variability modeling [4] which enables defining software product lines. The variability model is typically organized in a tree of features. A resolution of the tree structure constitutes a particular product. Defining the features [5] comprises a variability modeling (VM) process. There are several

approaches to variability modeling which make use of the feature concept, e.g. the cardinality-based feature modeling approach by Czarnecki et al. [6]. Pohl et al. [1] describe the Orthogonal Variability Model (OVM) methodology that prevents cluttering of a base language with variability concepts. The Common Variability Language (CVL) [7] exploits the feature term, defines variability orthogonally and specifies how to derive a concrete product [8, 9]. Fig. 1 sketches how CVL



(a) Containment (b) Component

(c) Generalization

Fig. 2: Containment, component and generalization (with placement and replacement fragments)

derives a product. The feature tree defines all possible configurations of a car, i.e. a car comprises an engine with two possible options (hp110, hp140), transmission (WD2, WD4) and some extra equipment (FrontSensor, RearSensor). We would like to have an engine with one hundred ten horsepower (hp110), two wheel drive transmission (WD2) and front sensor (FrontSensor) for our car; therefore, we need to choose the corresponding features. An engineer selects desired features defining the Resolution tree in CVL. We illustrate this definition by gray shading in the Feature tree, see Fig. 1.

To derive a product we specify how these abstract features are related to their concrete representations in a base model. In Fig. 2, there are three UML [10] class diagrams modeling the base model of a car. The derivation process is a set of substitutions which remove elements of a placement fragment and inject elements of a replacement. We have also defined two substitutions on our base model in Fig. 2. We replace the *Extras* class and corresponding containment (placement fragment, the diagram in Fig. 2a) with the *FrontSensor* class and associated containment relation (replacement fragment, the diagram in Fig. 2b). We also substitute *Unit* with the *Sensor* class to keep the generalization diagram in Fig. 2c consistent because *FrontSensor* is a *Sensor* specialization and not a *Unit* one. We do not show the necessary substitutions of the engine and transmission for the sake of simplicity.

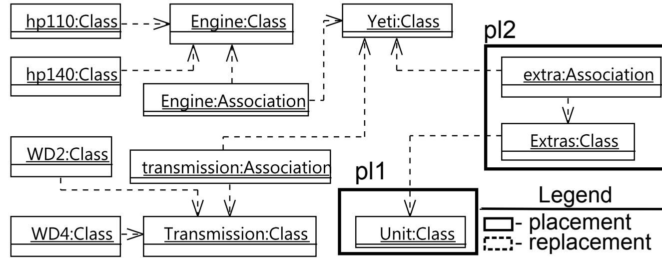


Fig. 3: Simplified instance diagram

Modern modeling languages such as UML may have quite large meta-models. A model in UML has a complex structure even for relatively small tasks. A diagram is a view of a specific model part. Entities in different diagrams may reference the same model elements. For example, we are not able to specify all modifying elements for the car base model as a single selection in one diagram since the involved classes and associations are present in different diagrams. A modeling practice shows that an engineer works only with one diagram at a time focusing on a specific part of a model.

In the instance diagram (see Fig. 3) of the UML meta-model [10], we show all components presented in Fig. 2. We use arrows to show links between objects. The UML meta-model specifies even more objects and links than shown in the figure. Fig. 3 outlines that even though the selections are made in different diagrams (containment and generalization) and may look completely independent, there are direct references from one placement to the other. We call such relations between fragments *adjacent*. In the given example, the result of the substitutions is well understood, i.e. we want a car with the front sensor which is a specialization of the *Sensor* class. However, if the substitutions are carried out independently as in MoSiS CVL [11, 12], the adjacent relation leads to an incorrect final product.

In the paper we explain our approach to the adjacent relation using an ABC example, present formal criteria and sketch algorithm to resolve this relation. Further, we apply the suggested approach on the presented motivation example to demonstrate the adjacent resolution technique. In addition, we categorize other problematic inter-placement relations and propose solutions to tackle them. We show that the graph rewriting techniques and tools [13–16] do not give us a necessary vehicle for conquering the presented challenges.

We organize the rest of the paper as follows. Section 2 covers background and related works. Section 4 gives a classification of the placement interferences using the ABC example for simplicity. In Section 5, we discuss the adjacent relation between placements while Section 6 elaborates all crossing cases. We walk through the introduced approach against the motivation example in Section 7. Finally, Section 8 concludes our work.

2 Background

2.1 Variability Realization in CVL

The basic concepts for a variability realization in CVL are placement fragment, replacement fragment and substitution combining them.

Definition 1 *Placement fragment is a set of elements forming a conceptual 'hole' in a base model, which may be replaced by a replacement fragment.*

Fig. 4 exemplifies a pair of the placement and replacement fragments. We highlight the placement by the solid oval line, while the dashed oval outlines the replacement. The elements inside ovals belong to the placement and replacement respectively. Placement and replacement fragments in CVL are defined via boundary elements depicted by black dots in Fig. 4.

Definition 2 *Boundaries are elements which represent the edges of a placement or replacement fragment.*

Definition 3 *ToBoundary is a boundary that represents a reference going from the outside to the inside of a placement or replacement fragment.*

Definition 4 *FromBoundary is a boundary that represents a reference going from the inside to the outside of a placement or replacement fragment.*

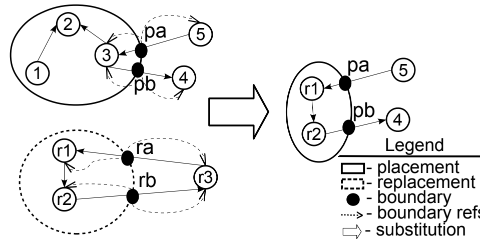


Fig. 4: Basic CVL concepts

A variability expert defines a placement or replacement using our tool through a simple selection procedure on a model. These elements inside placement or replacement may reference entities outside the given selection. Boundaries cut these references. Fig. 4 shows four boundary elements, i.e. two for the placement fragment (*pa* - toBoundary, *pb* - fromBoundary) and two for the replacement fragment (*ra* - toBoundary, *rb* - fromBoundary).

A subsequent execution of substitutions fragments modifies a base model deriving a new product in CVL.

Definition 5 *Fragment substitution is an operation that substitutes model fragment (placement fragment) for another (replacement fragment).*

The result of one fragment substitution is found in the rightmost of Fig. 4. The operation removes the placement elements and copy the contents of the replacement onto the recently cleared placement fragment.

Definition 6 *A binding is a map between a placement and replacement boundary elements.*

We specify that the element *5* should reference element *r1* and *r2* should point to *4* in the derived product via the binding of *pa* to *ra* and *pb* to *rb* respectively. Thus, bindings control substitutions instructing how replacement elements glue into existing structures.

3 Related Works

3.1 Conflicts in CVL

Oldevik et al. [17] analyze conflicts and confluence between substitution fragments in CVL. The paper states that transformations in CVL can be mapped to graph transformations in general case and checked using the critical pair analyzes. In our work we give more elaborated classification of interferences, check their confluence using graph transformation based tools, define solutions, formalize them and implement in the substitution engine [1].

Svendsen et al. [18] analyze conflicts in CVL on a Train Control Language (TCL) [19] example. They discuss two kinds of conflicts: border inconsistency and element inconsistency. The authors propose an algorithm to deal with inconsistencies in the base model by recording evaluation of the CVL model. Further, they use the original and evolution CVL models to derive a product. To perform the product derivation they analyze contextual information. Absence or shortage of the context may prevent an automatic product derivation. In our approach, we suggest evolving one CVL model and claim that the necessary information to derive a product automatically is always in the model. In addition, we illuminate other conflicts and propose solutions.

3.2 Confluence of Graph Transformations

Confluence of conflicting graph transformations plays a major role in the graph rewriting theory. Conflicts between transformations occur if transformations share common elements, the graph rewriting theory calls such transformation non-parallel independent. Heckel, Küster and Taentzer [16] give theoretical bases for identifying the parallel independence between transformations in terms of the rewriting theory. If two transformations are parallel independent then the local Church-Rosser theorem states that the transformations can be performed in any order yielding the same result [20]. Thus, we can speak of confluence in the parallel independent transformations. We do not consider confluence of

[1] One can find instructions to set up experiments at `http://goo.gl/9WD8Gx`

placement fragments without any relations between each other in this paper rather address cases where fragments are non-parallel independent (in terms of the graph rewriting theory). Confluence is also feasible for non-parallel independent transformations when all their critical pairs are confluent. A critical pair analysis of our motivation example reveals a non-confluent graph transformation system [21]. Therefore, the desired product is not possible to derive in the given settings. The basic graph approach is not capable of resolving the adjacent relation in general since the critical pair analysis reveals a non-confluent system.

3.3 Feature-Oriented and Delta-Oriented Programming

Feature-oriented programming (FOP) [22] is a step-wise refinement approach by Batory et al. [23] to the development of complex systems. A core idea of the step-wise refinement approach is that a product may emerge by adding features incrementally to a simple base model. Hence, we can avoid conflicts during a product derivation, which is different with respect to the CVL methodology of defining fragments. Batory et al. show that the approach can be applied to both code and non-code artifacts given that one defines the composition operation for each kind of artifacts.

Delta-oriented programming (DOP) [24] is an extension of the FOP paradigm and a novel programming language approach which operates with deltas to derive a product. Deltas allow removing elements from a product which is not generally allowed in the feature modeling. One may define a SPL on any language using the DOP paradigm. The approach proposes to resolve all conflicts between deltas by specifying the order of their resolution. The notion of deltas is somehow similar to fragments in CVL. However, one may define several fragments modifying the same elements in a model, which is a core distinction. Moreover, we consider substitutions as independent operations which the CVL engine may apply potentially in the arbitrary order. Therefore, the ordering is not a solution to conflicting fragments at least within the current CVL semantics. In addition, any specific resolution order of the adjacent fragments does not solve the problem with dangling references.

3.4 Aspect-Oriented Programming

Aspect-oriented Programming (AOP) is an approach to weave cross-cutting concerns into a program. Aspects are developed as separate units which can be applied independently. Lauret et al. [25] state that AOP suffers from a well-known composition issue i.e. several concerns are applied to the same join point. The problem is known as the aspect interference issue. Lauret et al. suggest inserting *executable assertions* to detect different kind of interference between aspects. As a solution to avoid undesirable interferences, the authors suggest ordering of conflicting advises. The notion of aspects is highly relevant to fragments in CVL which can be applied to the same model elements. However, the ordering of fragments to resolve conflicts is somewhat different with respect to CVL where

substitution operations do not have any particular order. In addition, ordering of substitutions does not help with adjacent fragments.

4 Placement Interference

4.1 Definitions and Concepts

A placement fragment forms a conceptual 'hole' in a base model according to Definition 1. The fragment substitution operation removes all elements of the placement creating a 'hole' in the model. Subsequently, the substitutions fills this 'hole' out with a copy of the replacement fragment. Any placement fragment is defined by means of boundary elements in CVL. Boundary elements reference objects outside and inside placement/replacement fragments (see Fig. 4) defining gluing points and elements to remove. Outside boundary references point to elements beyond a placement fragment. Boundary references outline also a set of affected elements (neighboring elements or gluing points), which we do not remove during a resolution process. We do not explicitly select these elements. Hence, we can conclude that a placement affects a set of objects which is wider than the set of the explicitly outlined objects by an engineer.

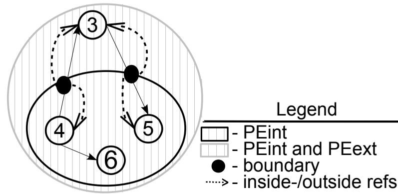
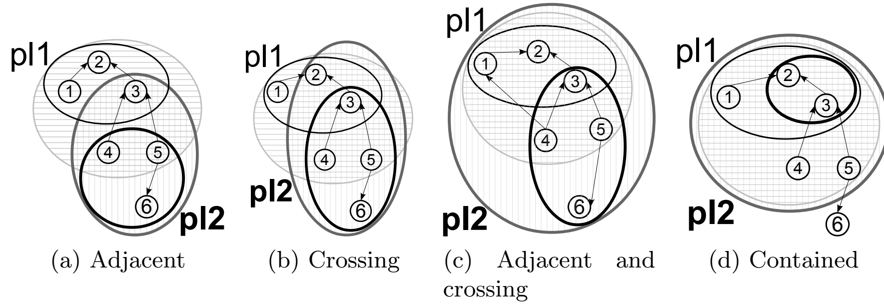


Fig. 5: Internal and external placement elements (PEint, PEext)

Definition 7 *Placement Element internal (PEint) is a set of all elements referred by inside boundary references and all elements in the transitive closure of all references from the elements in the set, but cut off at elements found through outside boundary references.*

Definition 8 *Placement Element external (PEext) is a set of all elements referred by outside boundary references.*

In Fig. 5 PEint = {4, 5, 6} and PEext = {3}. Thus, we define two sets of elements (PEint, PEext), which are affected by a selection. The dashed arrows pointing to *3* are *outside boundary* references, while the dashed arrows pointing to *4* and *5* are *inside boundary* references. The oval in Fig. 5 with the solid black border outlines PEint while the solid gray line highlights the union of PEint and PEext. Finally, we can conclude that PEint is a placement fragment in the CVL terminology, while PEext is a set of elements which relations are affected.



(a) Adjacent (b) Crossing (c) Adjacent and crossing (d) Contained

Fig. 6: Placement kinds of interference

4.2 Kinds of Interference

A variability engineer defines a set of elements to substitute via a selection in a base model. This selection is a set of objects which defines a placement fragment. This selection defines PEint that is a placement fragment and set of the affected elements, i.e. PEext. Thus, we can discuss relations between placements in terms of set relations. To find all possible relations between two different fragments we consider PEext $\bigcup$ PEint for each fragment and look for intersections between these unions. If the unions do not intersect then the substitution process goes smoothly. We do not discuss this case further. There are three unique intersection cases considering other combinations and simple 2x2 table. An overlap only between two PEext does not cause malformed configurations during resolution in MoSiS CVL [11]. Thus, we are left with two basic overlapping kinds. There is also a special case for an intersection between PEints, namely when one placement is fully contained by another placement. The given interference kinds are not mutually exclusive. Fig. 6 depicts four overlapping relations between placements which we will elaborate in the subsequent sections.

Definition 9 *Adjacent placements are placements, where $PEint_1$ intersects $PEext_2$.*

Definition 10 *Adjacent relation is a reference between two elements in different adjacent placements.*

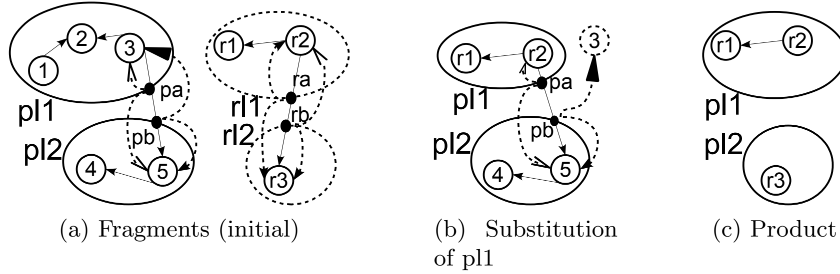
Definition 11 *Crossing placements are placements, where $PEint_1$ intersects $PEint_2$.*

Definition 12 *Crossing relation is a reference between two elements in different crossing placements.*

Fig. 6 shows three cases, where two placements conform to the definition of crossing placements.

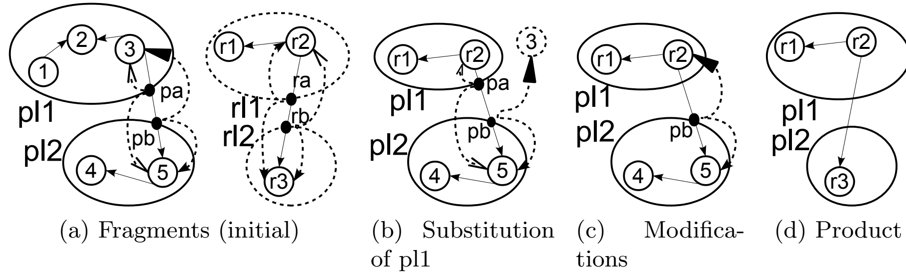
Definition 13 *Contained placements are placements, where $PEint_1 \subseteq PEint_2$.*

Definition 14 *Contained relation is a reference between two elements in different contained placements.*



(a) Fragments (initial) (b) Substitution of pl1 (c) Product

Fig. 7: Product (invalid) derivation without adjacent resolution



(a) Fragments (initial) (b) Substitution of pl1 (c) Modifications (d) Product

Fig. 8: Product (valid) derivation with adjacent resolution

5 Adjacent Placements

Independent substitutions of interfering placements cause dangling references in a variability model. Fig. 7 demonstrates a derivation process in MoSiS CVL [11]. There are two adjacent placements and corresponding replacements in Fig. 7a. We bind *pa* to *ra* and *pb* to *rb* to specify substitutions. A substitution of the adjacent fragment *pl1* results in a dangling *outside boundary* reference going from *pb* to the object *3* (an arrow with the filled large head) in Fig. 7b. Note, the object *3* is not in the model any more. A subsequent substitution of the second adjacent fragment *pl2* in MoSiS CVL yields an invalid product due to this reference, i.e. the product in Fig. 7c misses a link between *r2* and *r3*. Graph transformation tools (Henshin [26], EMorF [27]) consider the given transformations as non-confluent since the first substitution disables the second one. Thus, the desired substitutions are not feasible applying the graph techniques either. Fig. 8 sketches the derivation process with a necessary adjustment (see Fig. 8c) of the dangling reference to obtain the expected product in Fig. 8d.

The case shows that we cannot consider substitutions with adjacent placements as independent. We need to see these transformations together. A solution for the problem is to modify the variability model during an execution. In our example, if the *outside boundary* reference of *pb* (see Fig. 8b) pointed to the object *r2*, then the resolution would yield the proper model. Fig. 8c exemplifies

the required adjustment of the *outside boundary* reference. Thus, an essence of our approach to adjacent placements is to find and correct dangling references in such a way that they point to correct objects all the way through a derivation process. Summarizing, the adjacent resolution is a threefold process: 1) find adjacent placements, 2) find adjacent boundaries, 3) fix references of the adjacent boundaries during a product derivation.

Definition 9 gives necessary criteria to find adjacent placements, i.e. $\mathrm{PEint}_1 \bigcap \mathrm{PEext}_2 \neq \emptyset \wedge \mathrm{PEint}_1 \bigcap \mathrm{PEint}_2 = \emptyset$. Therefore, we need to walk through all placements in the model testing them against the proposed criterion. Two placements are adjacent placements if the criterion holds.

Further, we find all adjacent boundaries for adjacent placements. An adjacent relation between placements affects these boundaries yielding dangling references during substitutions. Thus, we have to modify them as the derivation progresses to keep the model consistent. Boundaries are adjacent if their *outside boundary* and *inside boundary* references match certain patterns. We formalize these patterns in Algorithm 1. Two adjacent boundaries are an adjacent boundary pair if these boundaries conform to the same match pattern. In Fig. 8a, the

```
Data: boundariesPlc1 - boundaries of the first adjacent placement,
      boundariesPlc2 - boundaries of the second adjacent placement
Result: adjBoundaryCurrent, adjBoundaryStale - boundary maps
for b1 ∈ boundariesPlc1 do
    for b2 ∈ boundariesPlc2 do
        if IsInstanceOf(b1) = FromBoundary and
        IsInstanceOf(b2) = ToBoundary then
            if b1.inside △ b2.outside = ∅ and b2.inside ⊆ b1.outside then
                adjBoundaryCurrent[b1] ← b2;
                adjBoundaryStale[b1] ← copyBoundary(b2);
            end
        end
        if IsInstanceOf(b1) = ToBoundary and
        IsInstanceOf(b2) = FromBoundary then
            if b1.outside △ b2.inside = ∅ and b1.inside ⊆ b2.outside then
                adjBoundaryCurrent[b1] ← b2;
                adjBoundaryStale[b1] ← copyBoundary(b2);
            end
        end
    end
end
```

Algorithm 1: Procedure to find adjacent boundaries

boundaries *pa* and *pb* constitute an adjacent boundary pair. Informally, an adjacent boundary pair is a pair of adjacent boundaries which cut the same adjacent relation.

Fig. 8c shows a modification we have to execute once we substitute *pl1*. We need to modify an adjacent boundary of the adjacent pair. The modification is a twofold process, i.e. 1) walk through adjacent boundaries of a not yet substituted placement removing pointers to invalid objects, e.g. the object *3* 2) correct boundary references to point to just replaced elements, e.g. object *r2* (see Fig. 8c). Algorithm 2 presents formally the outlined procedure. This procedure eliminates the dangling reference from the boundary *pb* to an element in the placement *pl1*.

Data: *boundaries* - boundaries of a not yet substituted adjacent placement; *adjBoundaryCurrent* - a map that stores adjacent boundary pairs and their current references; *adjBoundaryStale* - map that stores adjacent boundary pairs and their stale references (before substitution)
Result: fixed outside and inside boundary references
for $b \in boundaries$ **do**
    if $IsInstanceOf(b) = ToBoundary$ **then**
        $b.outside \leftarrow adjBoundaryCurrent[b].inside$;
    end
    if $IsInstanceOf(b) = FromBoundary$ **then**
        $b.outside \leftarrow b.outside \setminus adjBoundaryStale[b].inside \bigcup adjBoundaryCurrent[b].inside$;
    end
end

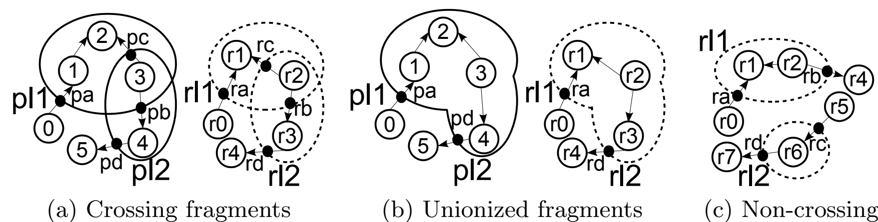
Algorithm 2: Fixing boundary references for adjacent placements

6 Kinds of Crossing Placements

6.1 Approach Overview

Fig. 6b, Fig. 6c and Fig. 6d outline all possible crossing kinds between placements. An engineer may define placement fragments in different diagrams that leads to crossing placements in the base model. An attempt to substitute these two placements one by one, results in dangling references and malformed products. In addition, two substitutions may replace elements of the crossing twice. This may cause different final products depending on the substitution order.

We argue that crossing placements should be considered as a single placement as well as their replacement fragments. Thus, we introduce a unionization procedure as a solution for this case. The crossing may originate from either a pragmatic need or an error in a variability definition. Therefore, we must be able to distinguish the cases. Required information for the decision is already in a variability model. By checking for unionizing crossing fragments we 1) spot erroneous variability definitions, tackle cases where the unionization operation is possible, 2) reduce the overall amount of substitutions facilitating the derivation process and 3) widen the semantics of the fragment definition which may enhance the variability specification process.



(a) Crossing fragments (b) Unionized fragments (c) Non-crossing

Fig. 9: Crossing fragments

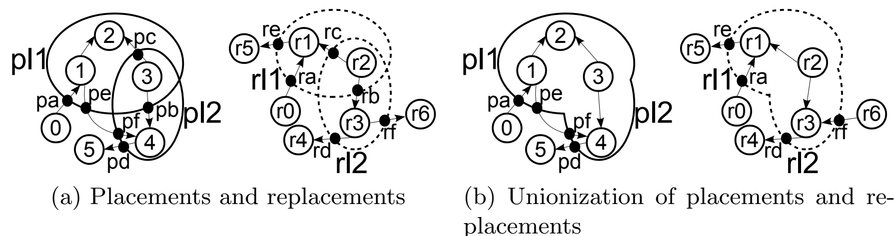
6.2 Crossing Placements

Two substitutions with crossing placements should be resolved as a single substitution, i.e. we should attempt to unionize the given placements and corresponding replacements. Boundary elements in CVL fully define placement and replacement fragments; therefore, we can alter boundaries in order to adjust fragments. The unionization of crossing fragments removes boundary elements which are internal to the unionized fragment. Boundary elements are removed when their *outside boundary* references point to elements inside the unionized placement since this contradicts the definition [7]. In Fig. 9a, we bind *pa* to *ra*, *pb* to *rb*, *pc* to *rc* and *pd* to *rd*. The boundary elements *pb* and *pc* are internal w.r.t. the unionized placement as well as *rb* and *rc*. Thus, we remove these boundaries to unionize the placement and corresponding replacement fragments. The unionization result is in Fig. 9b.

Let us now consider the same placement fragments from Fig. 9a and replacements in Fig. 9c. We bind the boundaries as in the previous case. The unionization approach suggests removing the placement boundary elements *pb*, *pc* and corresponding boundaries *rb*, *rc*. This unionizes the placement fragments. On the contrary, the replacement fragments become inconsistent and the remaining boundaries do not define a unionized replacement. In addition, the unionization of the non-crossing fragments does no make any sense. We consider an error in a variability definition when placement fragments overlap, but their replacements do not intersect or the replacement overlap of a different kind.

6.3 Crossing and Adjacent Placements

Fig. 10a shows an example of two placements and replacements that are both adjacent and crossing. We cannot apply to such fragments the adjacent technique since it does not handle the crossing relation. While the unionization procedure should eliminate the adjacent relation between two crossing placements. The boundary elements *pc* and *pb* define the overlap of the kind crossing placements, and *pe* and *pf* constitute the adjacent case. As in the crossing case, the boundary elements *pc*, *pb* and *rc*, *rb* have to be removed when we unionize two placements. These boundaries are internal w.r.t. the newly unionized placement and replacement. The reference in Fig. 10b (which creates the adjacent relation between



(a) Placements and replacements

(b) Unionization of placements and replacements

Fig. 10: Adjacent and crossing fragments

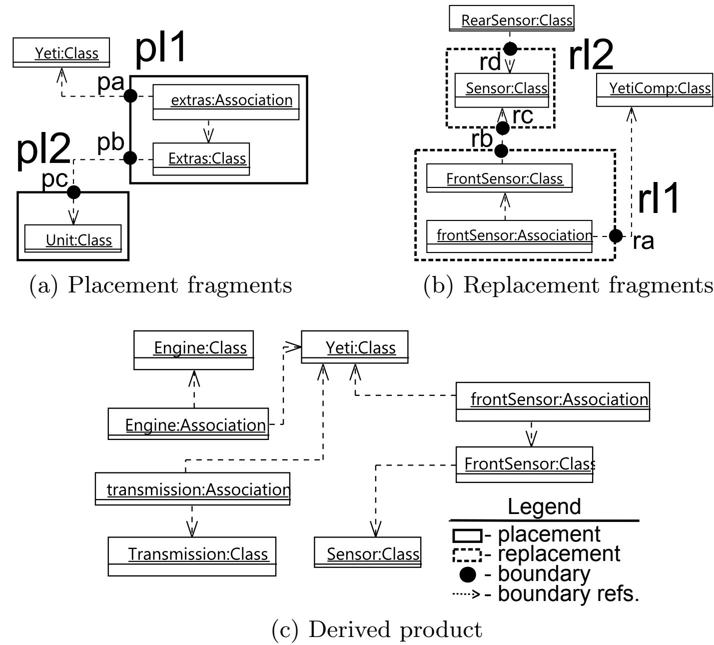
pl1 and *pl2*), is an internal link now. Hence, the adjacent relation is eliminated and there is no any interference . The pure crossing case is a special case of the crossing placements with the adjacent relation. Thus, developed criteria should be capable to handle both cases. The suggested unionization approach can tackle pure adjacent placements. However, unionization needs to consider relations between corresponding replacements. Thus, it reduces the amount of valid fragment definitions. The adjacent resolution method does not count on relations between replacement fragments. Therefore, the adjacent resolution method is more applicable to adjacent placements and can handle more cases than the unionization technique.

6.4 Contained Placements

An example of the contained placements in Fig. 6d. If we execute a containing placement, then the corresponding contained placement is never substituted. This resolution order never brings problems to the derivation process. On the contrary, a subsequent execution of the contained fragment and containing placement results in dangling references. We consider the contained placements as a potential problem in variability definition due to this ambiguity. The unionization procedure is also feasible for contained placements. We cannot find examples where such configuration is practically useful. Thus, we suggest unionizing for crossing fragments and reporting every time the substitution engine discovers contained placements.

7 Example Walkthrough

Let us finally walk through our motivation example from the introduction section. Fig. 11a and Fig. 11b depict two placement and replacement fragments which specify the desired transformation, i.e. we want to derive a car with a front sensor. In order to substitute *pl1* onto *rl1*, we bind *pa* to *ra*, *pb* to *rb*, while *pl2* substitution is achieved via binding of *pc* to *rc*, and *rd* is bound to *null* (we do not need this relation in the final model). We do not show inside-/outside boundary references just for the sake of neatness in the figures. The given placements in Fig. 11a are adjacent. Two adjacent boundaries *pb* and *pc* constitute



(a) Placement fragments (b) Replacement fragments

(c) Derived product

Fig. 11: Simplified instance diagram - walk through

the adjacent pair, i.e the *outside boundary* references of *pl1* and *pl2* point to the elements of the opposite placements. Hence, the outside references of these boundaries have to be modified during a transformation process.

Let us first substitute *pl1*, the outside boundary reference of *pc* points to *Extras:Class* which does not exist in the model; therefore, it is a broken reference. We know that *pc* is an adjacent boundary and should be modified to reference *FrontSensor:Class*. This reference is taken from *inside boundary* reference of *rc* which is bound to *pc*. Therefore, we are able to perform a substitution of *pl2*, where the substitution engine restores the link down to *Sensor:Class* from *FrontSensor:Class*. Two subsequent substitutions yield the product in Fig. 11c, which conforms to our expectations and definitions. We achieve the same result even if we perform substitutions in a different order.

We implemented the suggested approach in the CVL tool [1]. The procedure does not require any human interaction and completely automated as a derivation progresses.

8 Conclusion

CVL is a language to define software product lines. The language has the notion of fragments to specify elements to substitute in a model. Modern modeling languages may have complex meta-models; therefore, the tools, which implement

the corresponding meta-models, may use different diagrams to represent a model and facilitate the development process. Fragments defined in different diagrams may interfere in a model causing unintended results during a product derivation. A variability engineer can define interference intentionally, reflecting a pragmatic need to specify substitution fragments in different diagrams, or by accident where overlaps indicate a failure in a variability model. In this article, we classify the fragment interferences, i.e. adjacent, crossing, adjacent and crossing, contained placements. For each kind we define the detection criteria and how to handle them properly.

We have implemented the findings in the substitution engine developed at SINTEF [1] as well as demonstrated the proposed method to the adjacent relation on the motivation example. The engine performs substitutions, has functionality to detect and solve the adjacent relation. The resolution process of the adjacent relation includes the following steps: 1) detect adjacent relations, 2) find adjacent boundaries, 3) modify the adjacent boundaries. The engine executes the adjacent resolution procedure after a single substitution step to keep a variability model consistent all the way through a derivation process. The adjacent detection between placement fragments is a costly procedure. There are C_n^2 possible combinations, where n is the number of placement fragments and order is not important. Thus, we require to suggest optimizations to speed up this step. It is a part of our future work.

We have introduced the unionization approach to fragments with the crossing relation. There are three kinds of the crossing relation, i.e. crossing fragments, adjacent and crossing fragments, contained fragments. We demonstrate that the unionization approach is feasible only when placements and corresponding replacements have similar crossing kinds. Otherwise, a variability model is not consistent; thus, we assert an error. We argue that a case with contained placements indicates a potential problem in variability definition. The crossing resolution technique requires further elaboration and is not implemented in the engine yet which is a part of the future work.

Acknowledgment

The work has been carried out within the VARIES project [28]. We would like to thank our colleague at SINTEF for their valuable insights, as well as, anonymous reviewers and Maria Vasilevskaya at the Linköping University, Sweden for their helpful comments.

References

1. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. 1st edn. Springer Publishing Company, Incorporated (2010)
2. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages. Centrum voor Wiskunde en Informatika (2000)

3. Tolvanen, J.P., Kelly, S.: Integrating models with domain-specific modeling languages. In: Proceedings of the 10th Workshop on Domain-Specific Modeling. DSM '10, New York, NY, USA, ACM (2010) 10:1—-10:6
4. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P., Widen, T.: Consolidated Product Line Variability Modelling. In Käköla, T., Duenas, J.C., eds.: Software Product Lines. Springer Berlin Heidelberg (2006) 195–241
5. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (1990)
6. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: 11th International Software Product Line Conference (SPLC 2007), IEEE (September 2007) 23–34
7. OMG: Common Variability Language (CVL). OMG (2012) OMG document: ad/2012-08-05.
8. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: Software Product Line Conference, 2008. SPLC '08. 12th International. (2008) 139–148
9. Haugen, Ø., Wasowski, A., Czarnecki, K.: CVL: common variability language. In: Proceedings of the 16th International Software Product Line Conference - Volume 2. SPLC '12, New York, NY, USA, ACM (2012) 266–267
10. OMG: Unified modeling language: Superstructure (2005) version 2.0, formal/05-07-04.
11. Haugen, Ø.: CVL Tool from SINTEF (2010) Also available as http://www.omgwiki.org/variability/doku.php?id=cvl_tool_from_sintef.
12. Haugen, Ø., Wasowski, A., Czarnecki, K.: Cvl: common variability language. In Kishi, T., Jarzabek, S., Gnesi, S., eds.: SPLC, ACM (2013) 277
13. Heckel, R.: Graph Transformation in a Nutshell. Electronic Notes in Theoretical Computer Science **148**(1) (2006) 187–198
14. Pfaltz, J.L., Rosenfeld, A.: Web grammars. In: Proceedings of the 1st international joint conference on Artificial intelligence. IJCAI'69, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1969) 609–619
15. Pratt, T.W.: Pair grammars, graph languages and string-to-graph translations. J. Comput. Syst. Sci. **5**(6) (December 1971) 560–595
16. Heckel, R., Kster, J., Taentzer, G.: Confluence of typed attributed graph transformation systems. In Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G., eds.: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2002) 161–176
17. Oldevik, J., Haugen, Ø., Møller-Pedersen, B.: Confluence in Domain-Independent Product Line Transformations. In Chechik, M., Wirsing, M., eds.: Fundamental Approaches to Software Engineering. Volume 5503 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 34–48
18. Svendsen, A., Zhang, X., Haugen, Ø., Møller-Pedersen, B.: Towards evolution of generic variability models. In Kienzle, J., ed.: Models in Software Engineering. Volume 7167 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 53–67
19. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.J., Haugen, Ø.: The future of train signaling. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Model Driven Engineering Languages and Systems. Volume

5301 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 128–142
20. Ehrig, H.: Introduction to the algebraic theory of graph grammars (a survey). In Claus, V., Ehrig, H., Rozenberg, G., eds.: Graph-Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1979) 1–69
21. Vasilevskiy, A.: Conquering overlapping fragments in CVL. Master's thesis, University of Oslo (UiO) (pages 42-58, 2013)
22. Batory, D.: Feature-oriented programming and the ahead tool suite. In: Proceedings of the 26th International Conference on Software Engineering. ICSE '04, Washington, DC, USA, IEEE Computer Society (2004) 702–703
23. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. Software Engineering, IEEE Transactions on **30**(6) (June) 355–371
24. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In Bosch, J., Lee, J., eds.: Software Product Lines: Going Beyond. Volume 6287 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 77–91
25. Lauret, J., Waeselynck, H., Fabre, J.C.: Detection of interferences in aspect-oriented programs using executable assertions. In: Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on. (Nov.) 165–170
26. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place emf model transformations. In Petriu, D., Rouquette, N., Haugen, Ø., eds.: Model Driven Engineering Languages and Systems. Volume 6394 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 121–135
27. Klassen, L., Wagner, R.: Emorf-a tool for model transformations. Electronic Communications of the EASST **54** (2012)
28. ARTEMIS 2011, Project Number: 295397, VARIES: VARiability In safety critical Embedded Systems (2012-2015)